Escuela
Politécnica
Superior

Desarrollo de un motor de render 3D para Amstrad CPC 464

Grado en Ingeniería Multimedia

Trabajo Fin de Grado

Autor:
Joan Albert Sirvent Jerez

Tutor/es:
Francisco José Gallego Durán

Universitat d'Alacant
Universidad de Alicante

Septiembre 2017

1. Motivación y Objetivos

Cursar este grado y hacer el itinerario de videojuegos me ha reafirmado lo mucho que me gustan los videojuegos y programar.

He aprendido que actualmente se pueden desarrollar videojuegos de forma relativamente sencilla con herramientas muy avanzadas y potentes como Unity o Unreal Engine, pero también he aprendido cómo funcionan dichas herramientas por detrás durante el desarrollo del proyecto de 4º. De entre todos los aspectos que tocamos durante el desarrollo, uno de los que más complejo me pareció y más me fascinó fue el del renderizado 3D.

Además, la participación en el concurso CPCRetroDev fue muy estimulante y me enseñó la complejidad y la dificultad que existía para desarrollar videojuegos en máquinas tan limitadas como el Amstrad CPC 464, aunque el tiempo para el desarrollo del proyecto para dicho concurso fue bastante limitado y me quedé con ganas de realizar algo más complejo y desarrollado.

Es por todo ello que tras acabar el proyecto de 4º decidí proponerme un reto que, en caso de lograrlo, sería una buena manera de demostrar los conocimientos durante la carrera y mis capacidades como ingeniero.

El reto que me propuse, y que utilizo como proyecto final del grado, se trata del desarrollo de un motor de renderizado 3D para Amstrad CPC 464 y su integración en algún proyecto para mostrar su funcionamiento.

En general, el objetivo de este proyecto es el estudio de los motores 3D primitivos y el desarrollo de uno para Amstrad CPC 464 usando C y la librería CPCtelera, así como el desarrollo de un videojuego de ejemplo usando el motor de renderizado desarrollado.

En detalle, estos serán los objetivos a alcanzar en este proyecto:

- **El estudio de las limitaciones del Amstrad CPC 464,** para poder decidir y planificar como desarrollar el motor de renderizado.
- **El estudio de las técnicas de renderizado usadas en distintos videojuegos 3D,** seleccionando videojuegos desarrollados para máquinas con características similares al Amstrad CPC 464, resaltando los puntos fuertes y sus puntos débiles de cada técnica, así como la viabilidad de usarlas en una máquina como el Amstrad.
- **Implementar un motor de renderizado 3D en 8 bits para Amstrad CPC 464,** aplicando las técnicas y conocimientos obtenidos en el punto anterior para llevarlo a cabo.

- **Desarrollar un prototipo de videojuego** que muestre las capacidades del motor de renderizado desarrollado.

2. Agradecimientos

Agradezco en primer lugar a mi familia por el apoyo que me han dado y sobretodo a mi padre, que sin su apoyo y su dedicación durante todos estos años (y no han sido pocos) no me habría sido posible cursar y terminar esta carrera.

A mi tutor Fran, gracias a él descubrí el Amstrad y el concurso CPCRetroDev, y con él aprendí que para poder avanzar y aprender, primero hay que fallar. Además, compartimos la pasión por ese oscuro arte de la programación a bajo nivel.

A Miguel Ángel, por todo su apoyo, tanto durante este proyecto como durante los cursos anteriores.

A la Universidad de Alicante, por darme la oportunidad de estudiar esta carrera con un profesorado tan bueno.

Por último, agradecer a todos los amigos que he ido haciendo a lo largo de la carrera, por la ayuda prestada cuando ha hecho falta, por los buenos ratos que hemos tenido durante estos años y por todo el apoyo que me han prestado.

3. Índices

3.1.Índice de contenidos

3.2.Índice de figuras

4. Introducción

4.1.Terminología

En el documento se van a usar distintos términos, muchos de ellos abreviados para facilitar la lectura del documento.

- **Amstrad**: Amstrad CPC 464, ordenador de 8 bits comercializado a partir de 1984 y plataforma para la que se va a desarrollar el proyecto.
- **HW o Hardware**: componentes físicos del ordenador.
- **SW o Software**: procesos que se ejecutan en el ordenador.
- **CPU**: Central Processing Unit, hardware encargado de interpretar las instrucciones programadas.
- **RAM**: Random Access Memory, memoria sobre la que la CPU trabaja.
- **Frame**: Imagen mostrada por el ordenador en un momento dado.
- **FPS**: Frames per second, cantidad de imágenes que es capaz de mostrar el ordenador por segundo.
- **Bit**: Binary Digit, unidad mínima de información.
- **Byte**: Conjunto de 8 bits, conjunto mínimo de bits que se suele usar en programación.
- **Buffer**: Segmento de la RAM destinado a guardar datos para posterior uso.
- **Render o renderizado**: proceso de generación de una imagen a partir de datos interpretados por el ordenador.

Además, debido a la estructura de la CPU, se mostrarán valores y números en base 2 y en base 16, la simbología es la siguiente:

- **0x o #:** Los números que empiecen con 0x o # se interpretarán en base 16, además la cantidad de dígitos será múltiplo de 2 (0x0F, 0x010F, 0x1F00).
- **0b:** Los números que empiecen con 0b se interpretarán en base 2, además la cantidad de dígitos será generalmente 8 o múltiplo de 8 (0b00010000, 0b10101111).
- El resto de números se interpretarán en base 10.

Por ejemplo, 15 = 0x0F = 0b00001111.

4.2.Estructura del documento

El cuerpo del documento está dividido en 4 bloques diferenciados:

- **Amstrad CPC 464:** en este bloque se describen en detalle las características del ordenador para el que se va a desarrollar el proyecto y se enfatiza en aquellas que puedan beneficiar o lastrar el desarrollo del renderer.

- **Técnicas de renderizado 3D:** aquí se listan y describen distintas técnicas usadas para renderizado de imágenes 3D en máquinas de características similares al Amstrad y se nombran varios ejemplos reales de cada técnica.
- **Desarrollo del motor de renderizado:** este bloque describe el proceso de desarrollo del motor de renderizado, con las distintas iteraciones del desarrollo, los problemas encontrados y las soluciones aplicadas.
- **Desarrollo del juego:** por último, en este bloque se describe el proceso de desarrollo del juego de ejemplo y sus características.

5. Metodología

Puesto que se trata de un proyecto individual, no resulta necesario aplicar completamente una metodología de desarrollo, ya que éstas están enfocadas al desarrollo en equipo. No obstante, el uso de pautas seguidas por las **metodologías ágiles** sigue siendo útil a pesar de trabajar solo.

Concretamente la metodología en la que me he basado ha sido **Extreme Programming**, ya que permite ciclos muy cortos de desarrollo y pruebas, lo cual es necesario para poder probar los algoritmos implementados y modificarlos o sustituirlos completamente en caso de no ser factibles en ejecución.

De esta manera, el desarrollo se hace en ciclos muy cortos de implementación y testeo para poder comprobar que el algoritmo en desarrollo funciona correctamente o requiere de una remodelación o cambio total.

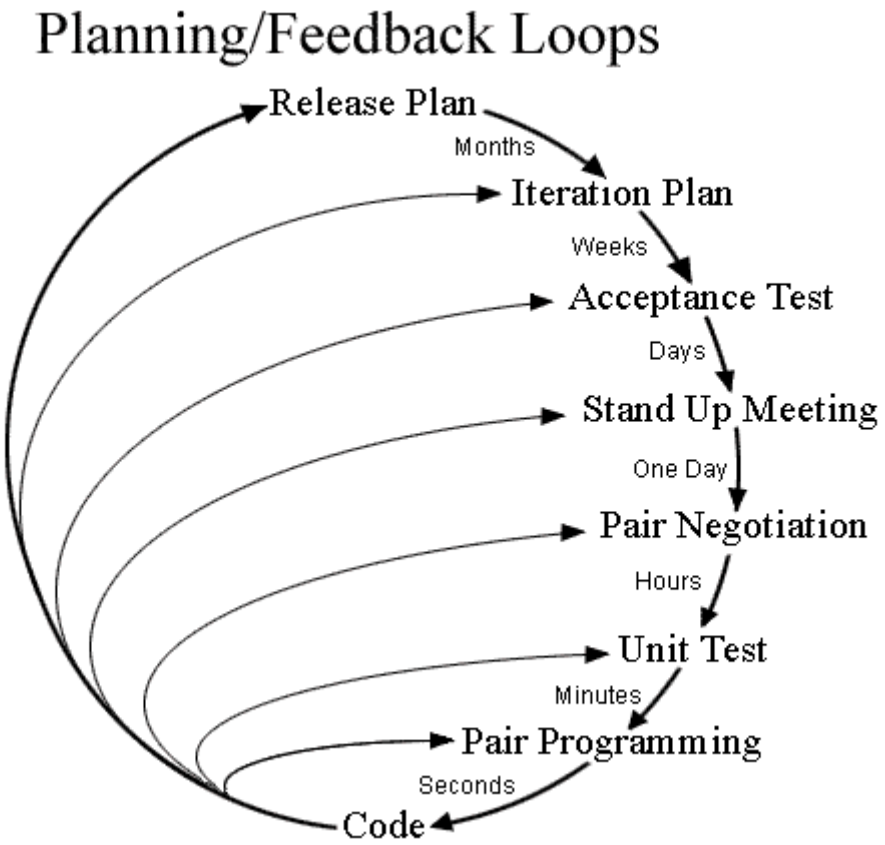

Figura 5-1. Extreme Programming

: en.wikipedia.org

Además, también se usará un sistema de control de versiones para evitar la pérdida de datos y para poder deshacer cambios en caso de ser necesario, en este caso se usará Git, ya que permite un buen manejo de ficheros de texto y comprobar el historial de ediciones, así como compara los cambios de manera sencilla.

Durante el desarrollo se van a utilizar las siguientes herramientas:

- **Notepad++:** Editor de texto con resaltado de sintaxis en C.
- **CPCtelera:** Librería de funciones en ensamblador y C para desarrollar en Amstrad.
- **Linux subsystem for Windows:** Subsistema de Linux en Windows 10, incluye el software necesario para compilar el proyecto.
- **Arkos Tracker:** Software de composición que permite exportar archivos de audio compatibles con Amstrad.
- **Gimp:** Herramienta open source de edición de imágenes, se usará para la creación de texturas del videojuego.
- **WinAPE:** Emulador de Amstrad, permite ejecutar software de Amstrad en los ordenadores actuales.
- **GeoGebra:** Herramienta matemática gráfica que permite visualizar funciones y geometría 2D.

6. Amstrad CPC 464

Para poner en contexto al lector y poder describir adecuadamente las características del Amstrad CPC 464, primero describiré de forma breve la situación de los ordenadores domésticos de la época.

A principios de la década de los 80 muchas compañías entraron en el negocio de la computación, esto llevó a una gran oferta de máquinas asequibles que empezaron a entrar en el ámbito doméstico. Los ordenadores domésticos habían llegado.

En esta primera oleada de ordenadores domésticos se encontraban máquinas como el Sinclair ZX en 1880, el ZX Spectrum y el Commodore 64 en 1982 o el Amstrad CPC 464 en 1984.

Todos estos ordenadores tenían en común que utilizaban el procesador Z80, un procesador de 8 bits muy extendido en la década de los 80 debido a su bajo precio y la facilidad de replicarlo, esto hizo que fuese el procesador por excelencia de la mayoría de ordenadores de la época.

El Amstrad CPC 464 se trata de un ordenador doméstico de 8 bits, con capacidad para mostrar 16 colores distintos de forma simultánea en pantalla. Cuenta con un lector de casetes para la carga de software y con una resolución máxima de 640 píxeles de ancho por 200 de alto.

6.1.Hardware

6.1.1.Procesador

El procesador del Amstrad, como se ha mencionado anteriormente, es un Zilog Z80 con una frecuencia de 4 MHz. Pero debido a que comparte memoria con el circuito de video, su frecuencia aproximada equivale a 3.3Mhz, ya que requiere sincronizarse con el reloj del chip gráfico para evitar conflictos.

Debido a que se trata de un procesador de 8 bits, trabajar con tipos de datos mayores a un byte, como enteros de 16 bits o números con coma flotante (32 bits), hará que reduzca considerablemente el rendimiento de renderer, por lo que se deben evitar el máximo posible.

6.1.2.Memoria

La memoria RAM del Amstrad consta de **4 bloques de 16 kilobytes** cada uno y es compartida entre la CPU y el chip gráfico. Esto implica que de los 64 KB totales, una parte se debe reservar para lo que se va a mostrar en pantalla.

Por defecto, la distribución de la memoria es la siguiente:

- **Primer bloque:** En los primeros 16 KB es donde se posiciona la ROM del firmware de Amstrad, una vez ha iniciado se puede elegir entre leer de la ROM o de la RAM, pero

no se puede leer del primer bloque de la RAM si está activa la ROM. Esto trae limitaciones como que si se está ejecutando código alojado en el primer bloque y se invoca la ROM no tendremos acceso al código que se estaba ejecutando y la aplicación tendrá un comportamiento no deseado.

- **Segundo bloque:** Este bloque por defecto está vacío, por lo que se puede usar libremente para almacenar datos y código.

- **Tercer bloque:** Este bloque contiene variables y funciones que utiliza el firmware de Amstrad para funcionar, se puede sobrescribir si se deshabilita el firmware. También se almacena en este bloque la pila de llamadas, que crece desde la posición más alta hacia la más baja.

- **Cuarto bloque:** Es el bloque usado por el chip de video para leer los datos que tiene que mostrar en pantalla. Desde la posición #C000 hasta la #FE80 son datos que se verán en pantalla, tras la memoria de video quedan 384 bytes sobrantes debido a que el chip gráfico usa un total de 16000 bytes.

6.1.3.Video

El chip gráfico de Amstrad permite distintas configuraciones y modos gráficos. Dispone de una paleta de 27 colores, de los que se pueden escoger una cantidad distinta dependiendo del modo gráfico.

6.1.3.1.Modos de texto

El Amstrad se puede usar en modo texto, en este modo permite 3 modos de texto distintos:

- 20x25 caracteres.
- 40x25 caracteres.
- 80x25 caracteres.

6.1.3.2.Modos gráficos

- **Modo 0:** Tiene una resolución de 160 píxeles de ancho por 200 de alto, la relación de aspecto de cada píxel es de 2:1 (coloquialmente conocido como píxel ladrillo por la forma que tiene al ser el doble de ancho que de alto). Este modo permite utilizar hasta 16 colores simultáneamente.
- **Modo 1:** Tiene una resolución de 320 píxeles de ancho por 200 de alto, la relación de aspecto es de 1:1. Permite usar hasta 4 colores simultáneos.

- **Modo 2:** Con una resolución de 640 pixeles de ancho por 200 de alto, este modo es el que más resolución permite, su relación de aspecto es 1:2. Permite utilizar únicamente 2 colores simultáneos.

Es importante destacar que todos los modos utilizan la misma cantidad de memoria, lo que cambia entre ellos es la forma de distribuirla, por lo que el tiempo que llevaría dibujar una pantalla completa es el mismo. El modo 0 sacrifica resolución para obtener una mayor gama de colores y el modo 2 hace lo contrario.

Teniendo en cuenta las características de cada modo, el modo que **se va a utilizar en el proyecto es el modo 0**, ya que permite mayor gama de colores y permitirá diferenciar mayor cantidad de elementos distintos en pantalla.

6.1.3.3.Interpretación de la RAM

El chip gráfico de Amstrad tiene una peculiar manera de interpretar los datos existentes en su zona de la memoria RAM. En lugar de dibujar líneas consecutivas, que correspondan a la memoria leída, **dibuja la octava línea posterior a la que acaba de dibujar**. Es por esto que cuando se escriba en la memoria de video hay que tener en cuenta que las imágenes se tienen que modificar para seguir dicho patrón, no se pueden escribir de forma lineal en la memoria de video.

Además de cómo se interpretan las líneas de video también hay que tener en cuenta cómo interpreta Amstrad los pixeles en cada modo gráfico y como se reparten en cada byte.

Para el **modo 0**, un byte almacena la información de 2 pixeles, **4 bits para cada pixel**, y se interpreta de la siguiente manera:

Mode 0, 160×200, 16 colors (each byte of video memory represents 2 pixels):

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
pixel 0 (bit 0)	pixel 1 (bit 0)	pixel 0 (bit 2)	pixel 1 (bit 2)	pixel 0 (bit 1)	pixel 1 (bit 1)	pixel 0 (bit 3)	pixel 1 (bit 3)

Figura 6-1. Interpretación de píxeles en modo 0

: www.cpcmania.com

Para el **modo 1**, un byte almacena la información de 4 pixeles, **2 bits para cada pixel**, y se interpreta de la siguiente manera:

Mode 1, 320×200, 4 colors (each byte of video memory represents 4 pixels):

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
pixel 0 (bit 1)	pixel 1 (bit 1)	pixel 2 (bit 1)	pixel 3 (bit 1)	pixel 0 (bit 0)	pixel 1(bit 0)	pixel 2 (bit 0)	pixel 3 (bit 0)

Figura 6-2. Interpretación de píxeles en modo 1

: www.cpcmania.com

Para el **modo 2**, un byte almacena la información de 8 píxeles, **1 bit para cada pixel**, en este modo cada bit del byte corresponde con el pixel del grupo de 8 pixeles de forma lineal.

7. Técnicas de renderizado 3D

Las primeras técnicas para mostrar imágenes 3D en videojuegos usaban pequeños trucos y engaños para simular mundos 3D. A continuación, hay listados algunos videojuegos que mostraban imágenes 3D del mundo que contenían y se van a comparar las características de la máquina sobre la que se ejecutaban con las características del Amstrad.

Están categorizados en función de si calculaban 3D real o simulaban el 3D.

7.1.3D simulado

Los primeros juegos que mostraron mundos 3D se ejecutaban en máquinas con poca potencia y cuyas características no permitían calcular la complejidad de un mundo 3D en tiempo real. Por ello se usaron trucos y técnicas que simulaban 3D.

7.1.1.Gráficos vectoriales

Los gráficos vectoriales tenían la ventaja de que solo requerían dibujar las líneas de los polígonos, esto permitía ahorrar muchos ciclos de cálculo a la hora de mostrar una escena. Sin embargo, la principal desventaja de este tipo de gráficos es que no permite dibujar superficies sólidas, únicamente las aristas de los objetos.

7.1.1.1.1.Battlezone (1980)

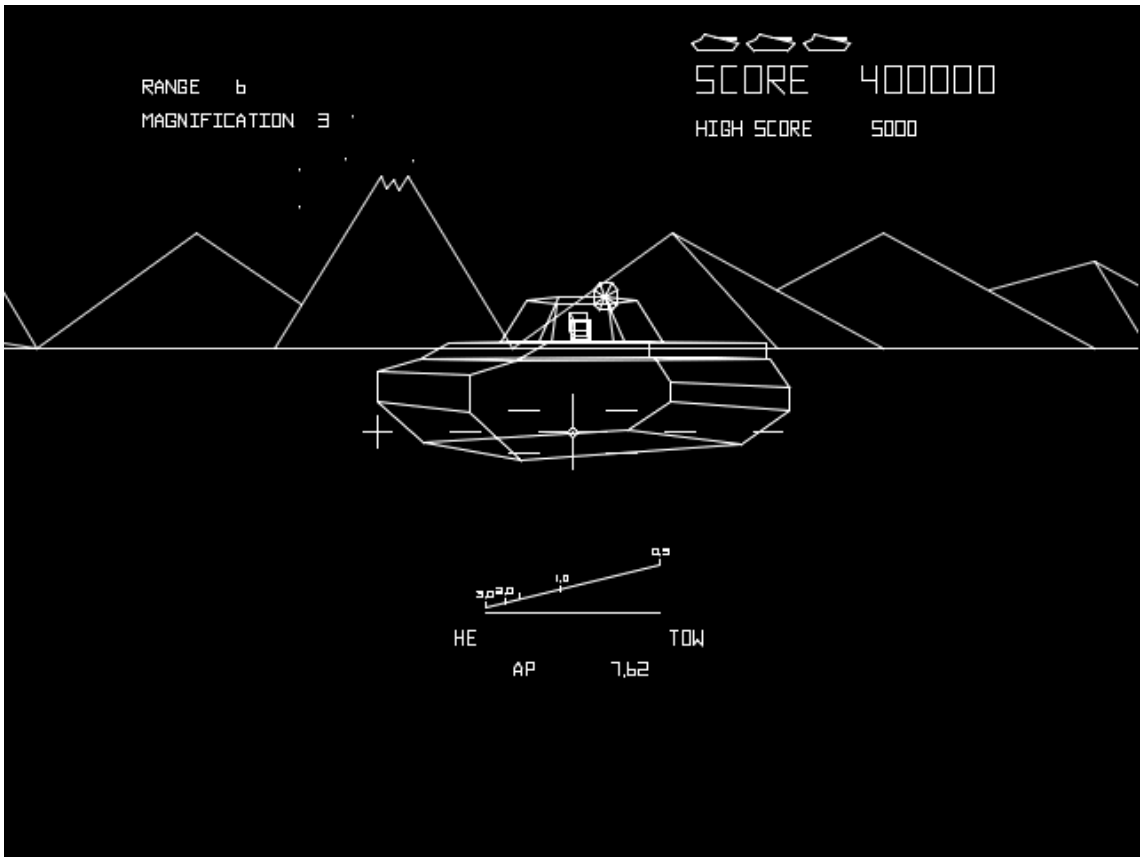

Figura 7-1. Captura del videojuego Battlezone
: en.wikipedia.org

Un ejemplo claro de este tipo de gráficos es el Battlezone, como se puede observar en la captura anterior, sus gráficos son muy básicos y consisten únicamente en líneas.

7.1.2.Gráficos rasterizados

Los gráficos rasterizados utilizan un ráster, o matriz de píxeles, para mostrar las imágenes, este es el método de dibujado que utiliza el Amstrad.

7.1.2.1.Raycasting

La técnica de raycasting consiste en lanzar un rayo desde la posición del observador para cada pixel de la pantalla y obtener la información de la colisión del rayo para determinar el color de dicho pixel. Los mapas se basan en una matriz de celdas 2D. Es por ello que no se considera 3D real, ya que no existen distintas alturas en los mundos de los juegos que usan esta técnica, sólo pueden recorrerse mapas planos.

7.1.2.1.1.Wayout (1982)

Figura 7-2. Captura del videojuego Wayout
: en.wikipedia.org

Se trata de un videojuego en primera persona desarrollado por Sirius Software y publicado para la Atari 8-bit, el Apple II y el Commodore 64.

En este juego el jugador debe recorrer uno de los 27 laberintos que contiene el juego y encontrar la salida con la ayuda de una brújula y un minimapa.

Este es un buen ejemplo de renderer, ya que las especificaciones de las plataformas para las que se publicó son similares a las del Amstrad.

Como se puede observar en la captura, los gráficos son muy rudimentarios, la vista 3D ocupa una fracción de la pantalla y no cuenta con ningún tipo de textura para los planos dibujados.

7.1.2.1.2.Wolfenstein 3D (1992)

Figura 7-3. Captura del videojuego Wolfenstein 3D
: en.wikipedia.org

Se trata de un juego de disparos en primera persona desarrollado por id Software para el sistema operativo MSDOS.

Aquí se puede observar la gran cantidad de elementos que está renderizando el juego en cada frame, además del texturizado de los elementos que aparecen.

Los requerimientos de este renderer están por encima de las capacidades del Amstrad, que por las fechas en las que se publicó este juego ya llevaba 8 años en el mercado.

7.1.2.2.BSP

Los **BSP o Partición binaria del espacio** es un método que subdivide de forma recursiva un espacio en elementos convexos. Esta subdivisión devuelve una estructura de datos que sirve para representar la escena y el mundo virtual del videojuego.

Con esta técnica se pueden calcular las salas y paredes que se van a renderizar, a diferencia del raycasting, permite tener distintas alturas en el mapa y no requiere que las paredes se posicionen en ángulos rectos entre ellas (ya que los mapas no se basan en matrices de celdas), no obstante, todavía no permite superposición de habitaciones y los mapas se siguen basando en información 2D.

7.1.2.2.1.Doom (1993)

Figura 7-4. Captura del videojuego Doom
: doom.wikia.com

Es un juego de disparos en primera persona desarrollado por id Software para el sistema operativo MSDOS.

Como se puede observar en la captura, muchas de las limitaciones que se tenían en los juegos que usaban la técnica de raycasting se han eliminado, como por ejemplo poder darle distintos ángulos a las paredes y distintas alturas al terreno.

Esta técnica de renderizado tiene unos requerimientos muy superiores a las capacidades del Amstrad.

7.2.3D real

A partir de 1996 las capacidades de los ordenadores domésticos y personales habían avanzado lo suficiente como para poder generar y calcular mundos 3D reales. Ya no se trataba de mapas 2D que se interpretaban como mundos 3D.

De esta manera se pudo avanzar considerablemente en el diseño de videojuegos, ya no se tenían las limitaciones de superposicionamiento de salas y se podían utilizar mallas poligonales para los objetos y entidades que saliesen en pantalla.

7.2.1.1.Quake (1996)

Figura 7-5. Captura del videojuego Quake
: quake.wikia.com

Desarrollado por id Software y publicado para MSDOS y Microsoft Windows.

Quake fue el primer videojuego con 3D real que se publicó.

8. Desarrollo del motor de renderizado

Para el desarrollo del motor de renderizado usé como base el algoritmo de raycasting utilizado en videojuegos como Wolfenstein 3D. Este algoritmo permitiría crear mapas con paredes sólidas y texturizadas.

El principal problema a abordar fue que el algoritmo está diseñado para ordenadores con una capacidad de cómputo mucho mayor que el Amstrad, por lo que tuve que realizar optimizaciones y recortes en sus características.

8.1.Maquetación inicial del algoritmo

Para empezar, decidí implementar de forma rápida el algoritmo en un motor de videojuegos actual (Unity), esto me permitiría comprobar que el algoritmo inicial da los resultados esperados y también me permitiría hacer modificaciones rápidas y comprobar resultados de manera más eficiente.

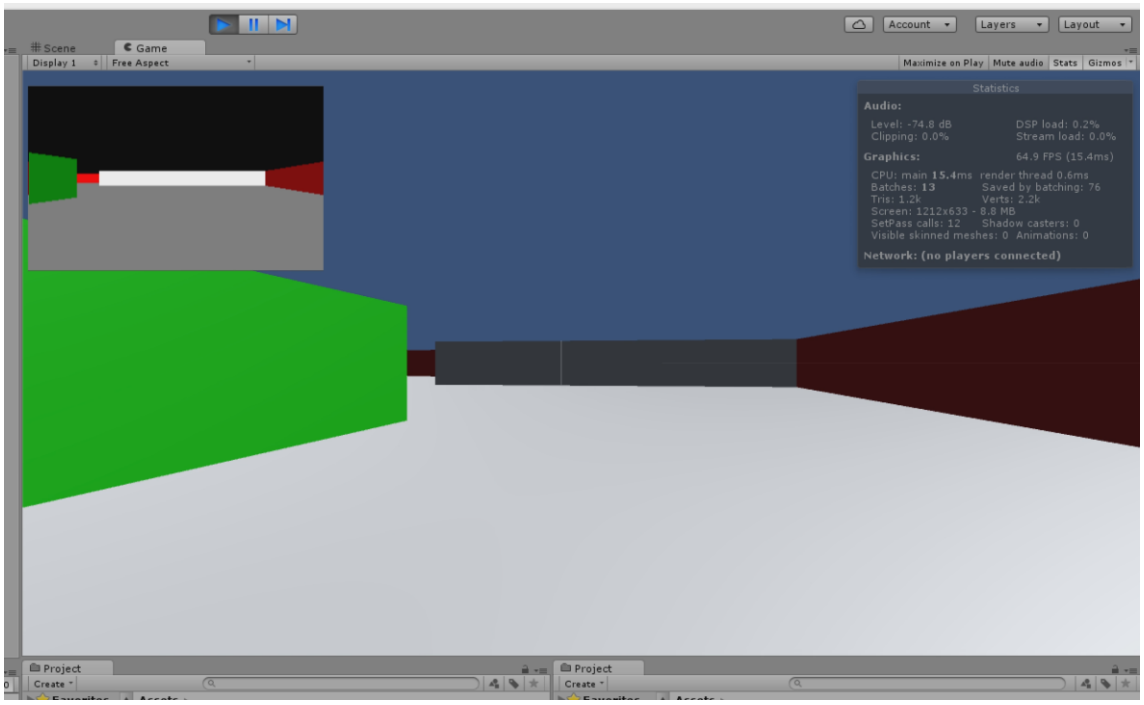

Figura 8-1. Captura del algoritmo de raycasting en Unity

Las bases y las referencias del algoritmo las obtuve de un tutorial escrito por Lode Vandevenne, en este tutorial explica las bases del renderer y su implementación.

Con ello hice la primera implementación en Unity, creando un renderer de raycast sin texturas.

El algoritmo funciona de la siguiente manera:

Para cada pixel de la pantalla, el algoritmo lanza un rayo desde la posición del jugador. La dirección del rayo se calcula en función de la obertura de la cámara y de la columna de píxeles que se esté dibujando.

```
rayDir.x = camera.transform.forward.x - ((camera.transform.forward.z *
 camPlaneMag) * currCamScale);
```

```
rayDir.y = camera.transform.forward.z + ((camera.transform.forward.x *
 camPlaneMag) * currCamScale);
```

En esta sección de código, la variable `camPlaneMag` corresponde con la obertura de la cámara, y `currCamScale` es la posición normalizada de la columna que se está dibujando actualmente. El resultado es la dirección del rayo en el plano XY, ya que no es necesario calcular en el eje Z porque el mapa es una matriz bidimensional.

A continuación, se calcula la distancia que recorre el rayo para alcanzar la siguiente celda, tanto en vertical como en horizontal.

```
float deltaDistX = Mathf.Sqrt(1 + (rayDir.y * rayDir.y) / (rayDir.y *
rayDir.x));
float deltaDistY = Mathf.Sqrt(1 + (rayDir.x * rayDir.x) / (rayDir.y *
rayDir.y));
```

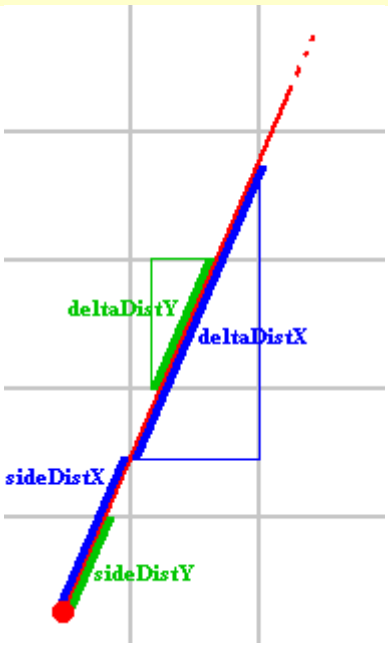

Figura 8-2. Descripción gráfica del raycast y las distancias entre celdas.
: lodev.org

Con esas distancias se va recorriendo la matriz paso a paso desde el punto de origen y comprobando en cada paso el valor de la celda.

En caso de que la celda contenga un valor correspondiente a una pared, se dibuja la columna actual con el color de la pared, teniendo en cuenta su distancia al origen.

Para calcular la altura en pixeles de la pared en la columna que se está dibujando se divide la altura del renderer (en pixeles) entre la distancia a la que está la pared y se centra en altura.

```
float lineHeight = wallDist < 1 ? h : h / (wallDist);
float startY = h / 2 - lineHeight / (2.0f);
```

Una vez calculada la altura y el color, se dibuja la columna de pixeles.

```
void verLine(int x, int y0, int y1, Color c)
    {
        for (int j = 0; j < y0; j++)
        {
            texture.SetPixel(x, j, Color.gray);
        }
        for (int j = y0; j < y1; j++)
        {
            texture.SetPixel(x, j, c);
        }

        for (int j = y1; j < h; j++)
        {
            texture.SetPixel(x, j, Color.black);
        }
    }
```

En este caso se está usando gris para el suelo y negro para el cielo, el parámetro de color que se pasa es el recogido por el rayo al colisionar con una celda de pared.

Como se puede observar, este algoritmo lanza únicamente un rayo por columna del renderizado, por lo que aumentar la resolución en horizontal hará que el renderizado sea mucho más costoso.

Una vez probado e implementado en Unity, adapté el algoritmo a C para probarlo en Amstrad.

8.2.Primera implementación

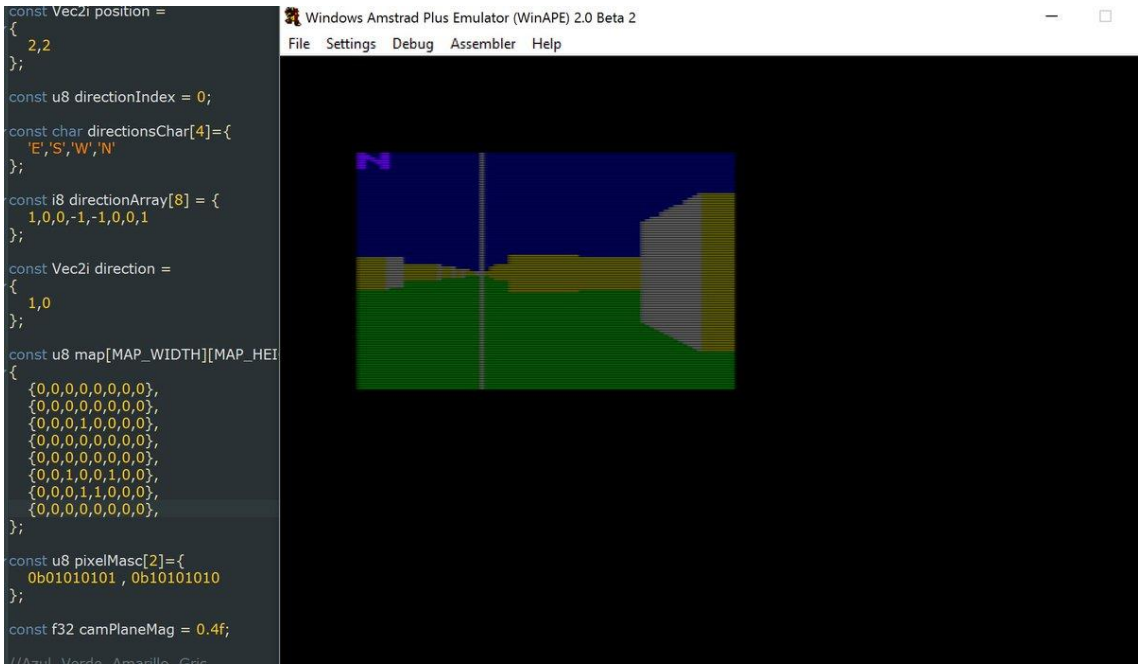

Figura 8-3. Captura del renderer de raycasting funcionando en Amstrad

La primera implementación del renderer fue una **adaptación directa del código** escrito para las pruebas de Unity.

Para mostrarlo en pantalla, el frame resultante del renderer se guardaba en un buffer en la posición de memoria 0x0040.

Con una resolución de 80 píxeles de ancho por 100 de alto en modo 0, la imagen resultante del renderer era de 4000 bytes.

Para mostrar ese resultado en pantalla implementé una función que leía 4000 bytes a partir de la posición 0x0040 y los copiaba en la memoria de video adaptándolos para que el chip gráfico los interpretara de forma correcta.

```
void Render(){
    u8 x ,y;
    u8* lineMem;
    u8* linePos = (u8*)SCREEN_TEXTURE_BUFFER;
    for(y=0;y<SCREEN_TEXTURE_HEIGHT;++y){
        lineMem = CPCT_VMEM_START + ((y&7) *0x0800) + ((y>>3)*0x0050);
        for(x=0;x<SCREEN_TEXTURE_WIDTH_BYTES;++x){

            *(u8*)lineMem = *(u8*)linePos;
            ++lineMem;
            ++linePos;
        }
    }
    cpct_drawCharM0(CPCT_VMEM_START,
5,1,directionsChar[directionIndex>>1]);
}
```

La función calcula la posición de memoria inicial de cada fila en la memoria de video y copia de forma secuencia los bytes.

Como se puede observar en la imagen que encabeza esta sección, el **algoritmo funciona y renderiza** mediante raytracing el mapa de ejemplo usado.

Sin embargo, existen varios **problemas importantes con este algoritmo**:

- El uso de tipos de datos mayores a 1 byte: el algoritmo usa muchas variables de 16 bits y de coma flotante de 32 bits, esto hace que el rendimiento caiga drásticamente en el Amstrad, ya que el procesador es de 8 bits.
- El rendimiento general del renderer es pésimo, consigue dibujar 1 frame cada 10 segundos. Por lo que es totalmente inviable para usar en un videojuego.
- Por último, se trata del renderer más básico, solo renderiza con colores planos, por lo que no se pueden meter detalles en las paredes como se podría hacer con texturas.

Debido a estos problemas decidí rediseñar el algoritmo manteniendo algunas características que tenía este.

8.3.Rediseño del algoritmo y maquetación

Para el rediseño del algoritmo decidí fijar que limitaciones le iba a imponer al renderer y que requisitos debería tener.

Para empezar, el renderer debería **funcionar a una velocidad suficiente** como para que el juego no se volviera aburrido. No debería ser necesario que renderizase para juegos a tiempo real, con que sirviera para juegos por turnos sería suficiente.

Segundo, el renderer debería poder **dibujar texturas** en las paredes para poder darle más calidad a los escenarios. Esto implica que además de calcular la distancia para cada columna de render, también se debe calcular la altura para la textura.

Tercero, para mejorar el rendimiento y poder optimizar, el renderer debería estar **fijado a 4 direcciones** correspondientes con los puntos cardinales, así como a una posición por celda del

mapa. De esta manera no es necesario usar números con coma flotante y se pueden hacer los cálculos con enteros.

Con estos requisitos y limitaciones me di cuenta de que se podía calcular las paredes que tenía delante y dibujarlas sin necesidad de lanzar rayos.

Únicamente necesitaba listar de forma ordenada las celdas que había en el cono de visión del personaje y fijar un valor de reducción del tamaño por distancia.

Para maquetar el renderer utilicé GeoGebra debido a que permite crear geometría 2D y visualizarla de forma sencilla y realizar cálculos geométricos rápidamente.

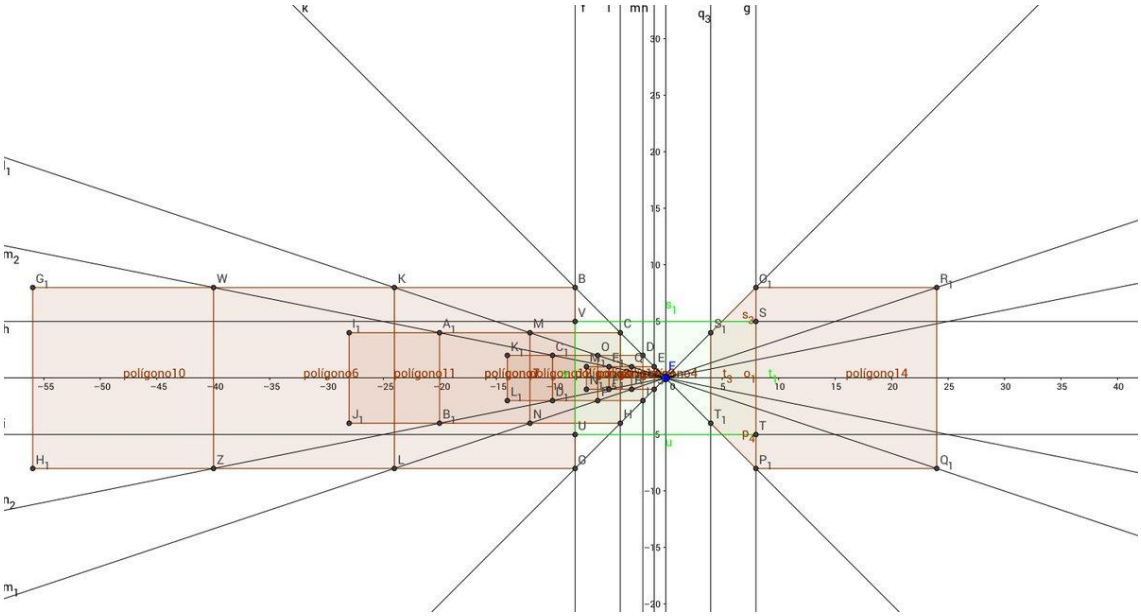

Figura 8-4. Maquetación en GeoGebra

Usando GeoGebra visualicé rápidamente las proporciones de las filas de celdas y la profundidad máxima de filas que deberían aparecer en el renderer.

Fijé la profundidad máxima en 5 filas desde la posición del personaje hacia el horizonte y la reducción en tamaño de una fila a la siguiente en ½ (cada fila consecutiva se ve ½ de alta que la anterior).

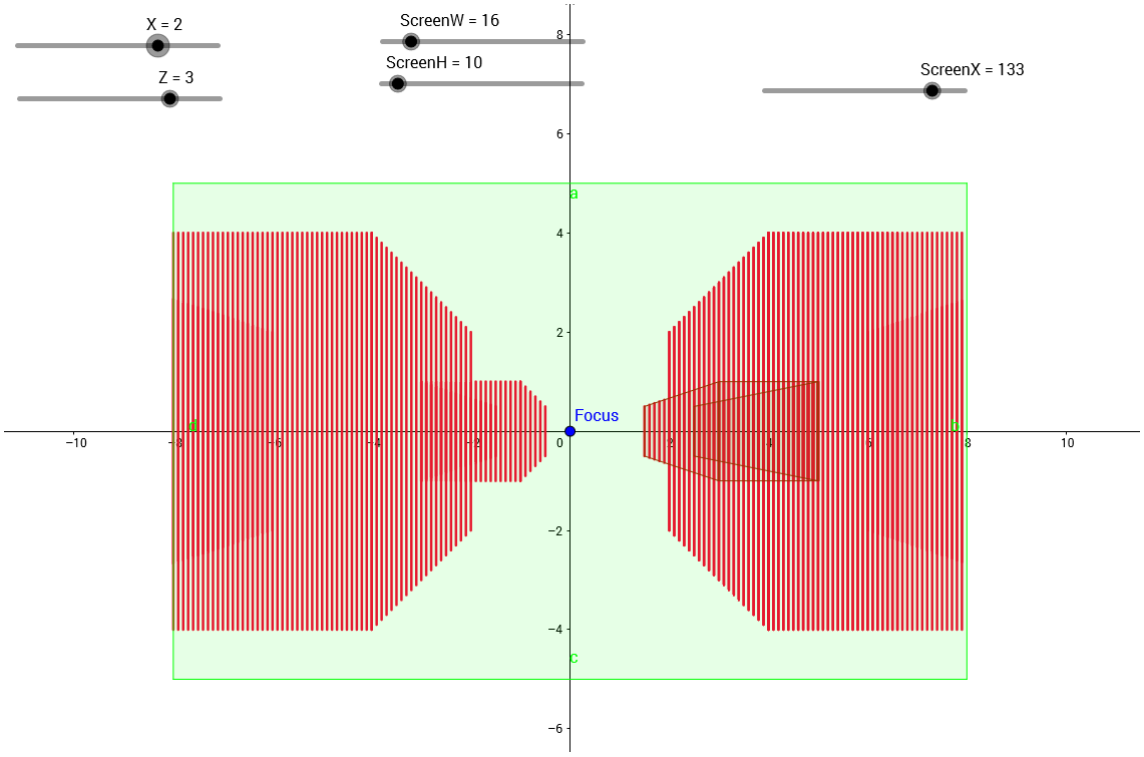

Figura 8-5. Resultado final del maquetado en GeoGebra

El resultado final del maquetado permite observar una simulación del renderer y comprobar que las proporciones son adecuadas.

Cada “bloque” de paredes en el maquetado corresponde a unas coordenadas (X y Z en la captura) que corresponden a su posición horizontal en la pantalla y su profundidad en filas.

8.4.Segunda implementación

Basándome en la maquetación realizada en GeoGebra, inicié la implementación en C del renderer.

El renderer funcionaría sobre un mapa plano, basado en una matriz bidimensional de celdas donde cada celda tiene un identificador de suelo o pared.

Lo primero que necesitaba era una función que me crease un array de las celdas que hay dentro del cono de visión del personaje.

El cono de visión del personaje es estático en tamaño, únicamente cambia la orientación en ángulos de 90º, por lo que es sencillo obtener las celdas que entran dentro.

Usando la maquetación realizada anteriormente se pueden obtener las medidas del cono de visión, siendo las siguientes en cantidad de celdas por fila y de más lejos a más cerca: 33,17,9,5,3,1.

Como se puede observar, la cantidad de celdas por fila sigue un patrón:

```
nActual = (nAnterior-1)/2 + 1;
```

Además, a esos valores hay que sumar 2, por las celdas que, a pesar de quedar fuera por cada extremo, se les puede ver la pared lateral. También se debe tener en cuenta que las celdas están

centradas en el cono de visión, por lo que al seleccionarlas de la matriz del mapa hay que tener en cuenta el offset que tienen delante.

Es por ello que para implementar la función que devuelve el array de celdas en el cono de visión, he usado una lista de offsets que sirven a la vez para calcular la cantidad de celdas de cada fila que hay en el cono de visión.

```
const u8 offsets_cells_in_view[5]={
    8,12,14,15,16
};
```

Ese offset se usa en un bucle que se encarga de añadir las celdas de cada fila al array:

```
for(i=offset;i<35-offset;++i)
```

Con esta implementación el valor i va a recorrer la cantidad necesaria con el offset necesario en cada fila produciendo el siguiente cono de visión:

```
********************************* * = celdas dentro de visión
........********************........ · = celdas fuera de visión
............**********............
..............*******..............
................*****................
..................***..................
                   P
```

Esa misma función ya se encarga de tener en cuenta la dirección en la que está mirando el personaje y rotar el cono de visión hacia dicha dirección.

Una vez obtenido el array de celdas a dibujar, el dibujado es sencillo. Se recorre el array y **se dibujan las celdas de lejos a cerca** teniendo en cuenta las proporciones obtenidas durante la maquetación.

Dibujar de lejos a cerca permite incluir una característica al renderer que con la técnica de raycasting no era posible incluir: **se pueden usar transparencias en las paredes** para poder ver que hay detrás de ellas.

Figura 8-6. Ejemplo de transparencia con el renderer realizado

También el hecho de **dibujar de lejos a cerca conlleva que el renderer tenga que dibujar 6 veces** para generar un frame, una vez por cada fila dibujada. Pero a pesar de ello, el rendimiento respecto al raycast es mayor, debido principalmente a la **ausencia de tipos de datos mayores a un byte**. La función de renderizado funciona completamente con enteros de 1 byte, por lo que el procesador solo necesita un ciclo para cada operación que realice.

El funcionamiento del renderer es el siguiente:

Empieza por la fila más alejada del personaje y calcula la cantidad de pixeles en ancho que ocupa una celda.

A continuación, empieza a dibujar usando el array de celdas que existen en dentro del cono de visión previamente calculado.

Siempre tiene en cuenta si la celda anterior tenia pared o estaba vacía para tener en cuenta si tiene que dibujar alguna pared lateral en los laterales de la celda actual.

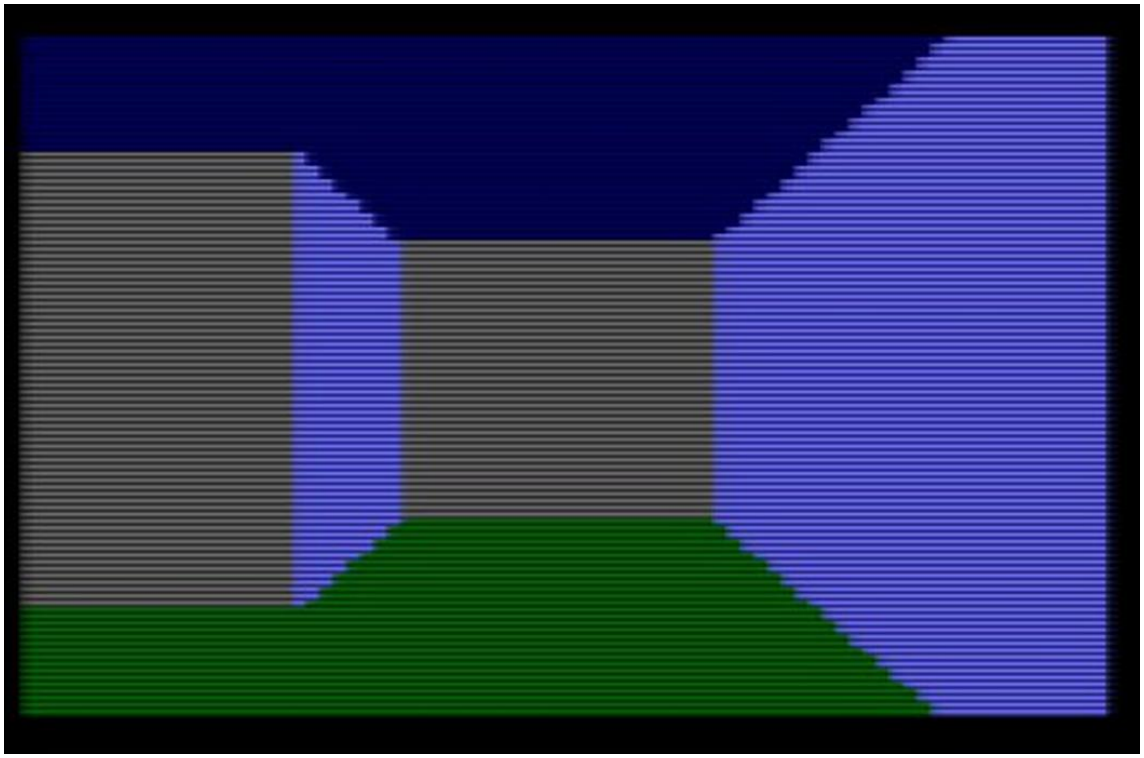

Figura 8-7. Paredes laterales

La imagen anterior ilustra la situación, en el centro se puede observar una celda vacía rodeada de celdas con paredes. El renderer va de izquierda a derecha dibujando y siempre tiene en cuenta si la anterior celda fue pared para dibujar los laterales, que en este caso son azules.

Cuando ha acabado de dibujar una fila, recalcula el tamaño en pixeles de cada celda para la nueva fila y vuelve a empezar de izquierda a derecha. Lo repite hasta acabar la fila que el personaje tiene delante.

Para cada columna de pixeles utiliza el ancho de cada celda para la altura. Y para dibujar la columna se usa la misma función que con el renderer de raycasting, se dibujan el suelo y el cielo y se dibuja de arriba hacia abajo la parte que corresponde a la celda.

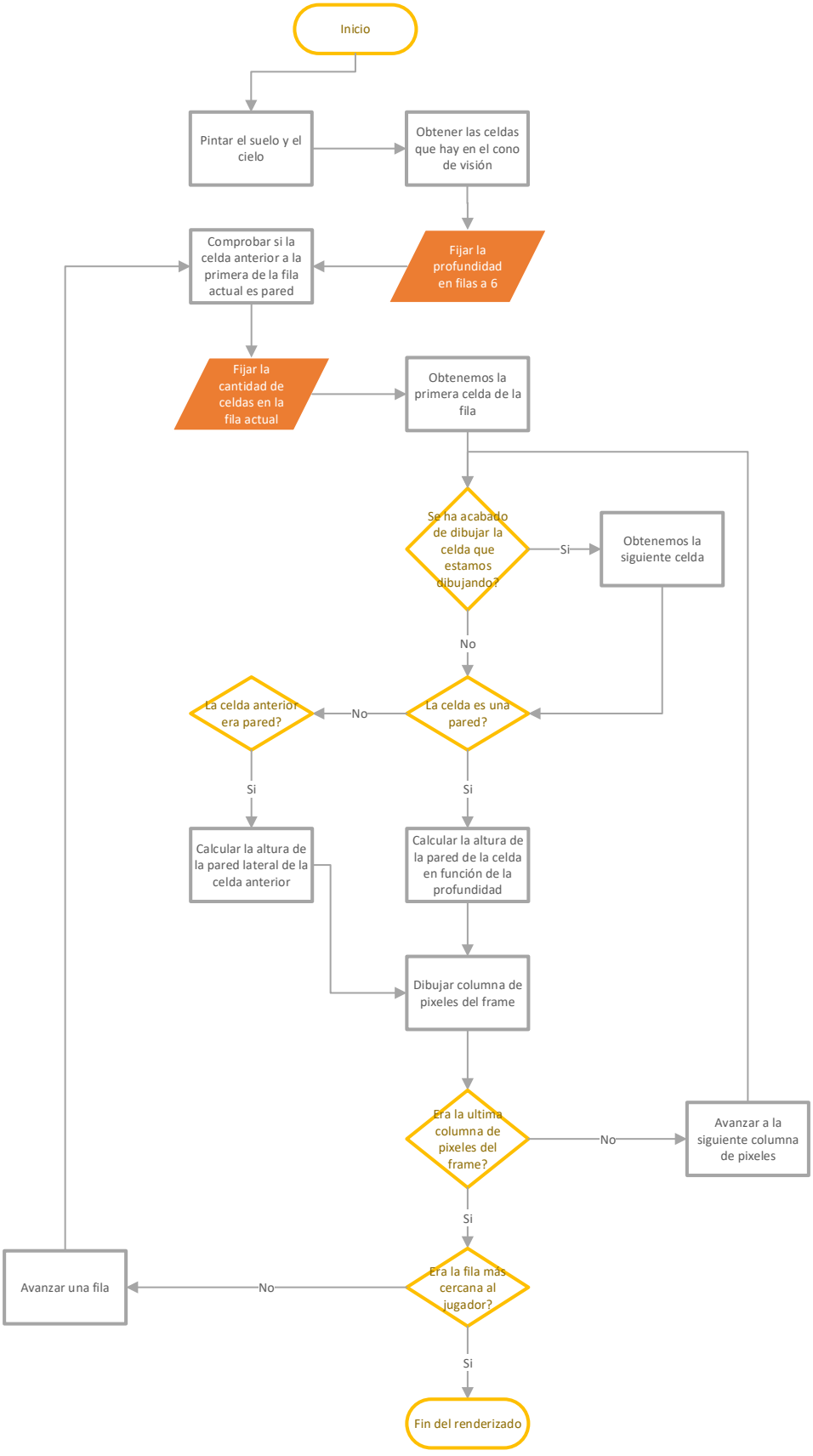

Figura 8-2. Esquema del funcionamiento del renderer

8.5.Texturizado

El siguiente paso fue incluir texturizado a las paredes.

Para ello decidí usar **texturas de 32x32 pixeles**, puesto que con esa resolución ya se pueden conseguir detalles interesantes en las paredes.

Para el texturizado, modifiqué la función encargada de dibujar cada columna del frame para que dibujase un color extraído de la textura asociada a la pared en lugar de dibujar un color plano en la zona correspondiente de dicha la pared.

El mapeado de texturas es muy sencillo, cada celda tiene un ID asociado a una textura, por lo que todas las paredes de dicha celda tendrán la misma textura asociada.

Para obtener el color durante el renderizado, la función de dibujado de columna obtiene un parámetro que le indica la columna de la textura actual y la fila la calcula usando la altura en pixeles de la pared para esta columna del frame. Sabiendo la altura de la pared y de la textura se calcula para cada pixel del frame que color de la textura usar.

Para evitar usar números con coma flotante, en lugar de calcular la nueva posición en cada pixel, se calcula la diferencia que hay que sumar y se hace un cálculo acumulativo, de manera que cada vez que se dibuja un pixel se suma una cantidad entera a la posición anterior.

Para la transparencia se usa un color especial, el magenta. Todos los pixeles de color **magenta se tratan como transparentes** y, por lo tanto, no se dibujan.

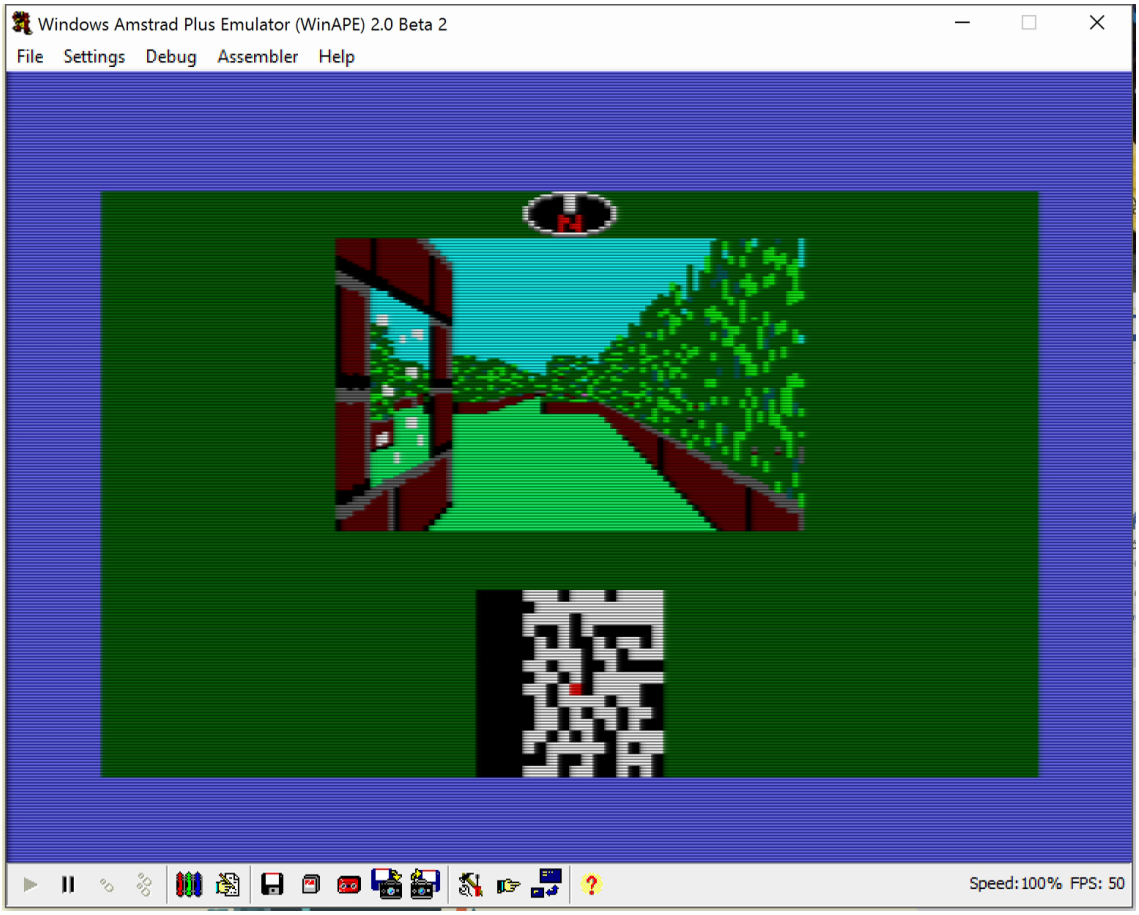

Figura 8-8. Captura del renderer con texturizado de paredes.

El siguiente paso es incluir entidades distintas a paredes y que el renderer pueda dibujarlas.

8.6.Inclusión de entidades

Para la inclusión de entidades distintas a paredes en el renderer simplemente se tiene que añadir más identificadores que pueda interpretar el renderer.

La principal diferencia entre las entidades y las celdas con paredes, es que las entidades no tienen profundidad, por lo que el renderer no necesita tener en cuenta si la celda anterior tenía una entidad para dibujar el lateral.

Además de eso, implementé 2 funciones distintas para los distintos tipos de entidades debido a que sus texturas tienen un tamaño distinto al de las paredes. Esto se podría haber hecho modificando la función original de dibujado para que admitiera distintos tamaños, pero eso acarrearía un mayor coste computacional para el dibujado, y siendo una función que se ejecuta múltiples veces durante el dibujado de un frame es conveniente reducir el coste computacional a costa de mayor tamaño en memoria.

Las entidades se dividen en 2 tipos, objetos y enemigos, cada tipo tiene un tamaño de textura distinto:

- Los objetos usan texturas de 16x16 pixeles.
- Los enemigos usan texturas de 24x24 pixeles.

A pesar de tener un tamaño de textura menor, la función de dibujado sigue considerando el ancho de la celda de 32 pixeles, por lo que no empezará a dibujar a la entidad hasta pasados unos pixeles. De esta manera se consigue que las entidades sean más pequeñas que las paredes y no ocupen todo el ancho de la celda.

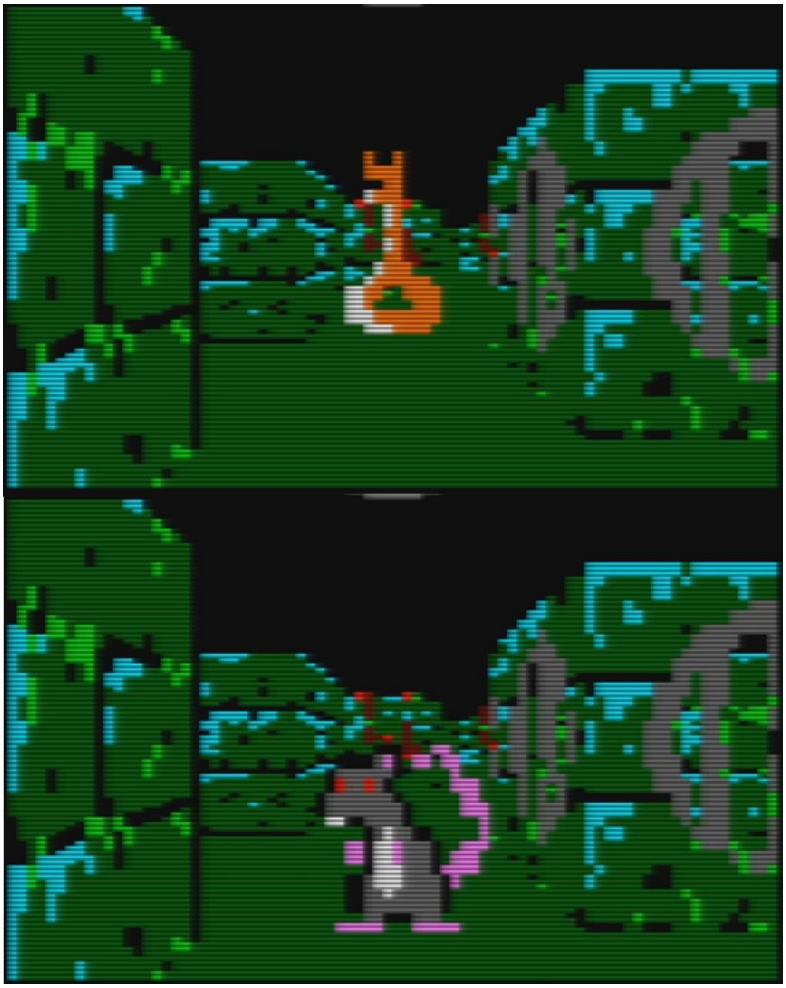

Figura 8-9. Ejemplos de entidades

8.7.Resultado final del renderer

El resultado final del renderer es bastante satisfactorio. La velocidad de renderizado final está entre 1 y 2 FPS, suficiente como para poder usarlo en videojuegos por turnos.

8.7.1.Características

La implementación final del renderer tiene características interesantes que no se podían conseguir usando un renderer de raycasting en una máquina con estas características, como la transparencia o la texturización de las paredes.

8.7.2.Limitaciones

Las limitaciones del renderer son notorias, no permite tener libertad de movimiento ni de rotación, el personaje está limitado a moverse de celda en celda y rotar en intervalos de 90º. Además, debido al limitado framerate, limita el tipo de juegos que lo pueden usar a ser por turnos.

9. Juego de ejemplo

Para demostrar el funcionamiento del renderer creado, desarrollé un videojuego que usase todas sus características.

9.1.Diseño

Teniendo en cuenta las características y limitaciones del renderer, debía hacer un juego por turnos y hacer uso de objetos y enemigos.

Un dungeon crawler es perfecto para las características del renderer. Los dungeon crawler son un tipo de juego en el que el jugador debe recorrer una mazmorra (normalmente generada aleatoriamente) buscando objetos y enfrentándose a enemigos para poder avanzar.

Puesto que el juego iba a servir como demostración de las características del motor, diseñé todo entorno a dichas características.

Los mapas del juego se crean a partir de un algoritmo aleatorio, de manera que cada partida que se juega es distinta. Además, una partida completa consta de 32 niveles distintos divididos en 4 zonas temáticas distintas, cada zona usa un conjunto de texturas y colores para el mapa y los enemigos.

El juego también tiene un sistema de guardado y cargado de partidas, para evitar tener que jugar los 32 niveles seguidos.

El diseño de las texturas y sprites, se encargó de realizarlo Alejandro Padilla Lozoya. Adecuándolos a las limitaciones técnicas del renderer y del Amstrad.

La música la compuso Carlos Blaya Cases, usando el software Arkos Tracker.

9.2.Estructura del motor del juego

El motor del juego es una máquina de estados, cada estado representa una parte distinta del videojuego.

La mayor parte de los estados son los distintos menús del juego, además también hay un estado para la pantalla de carga y un estado “en partida”.

Todos los estados comparten una misma estructura y el motor solo tiene que conocerla para poder llamar a las funciones de cualquier estado, de esta manera se facilita la creación e inclusión de nuevos estados.

Todo el juego está dirigido por eventos de menús, incluso dentro de la partida el jugador maneja el juego seleccionando elementos del menú de la interfaz, de esta manera se evita que haga acciones inintencionadas en caso de que una acción tarde más de lo debido.

9.3.Algoritmo de generación de mapas

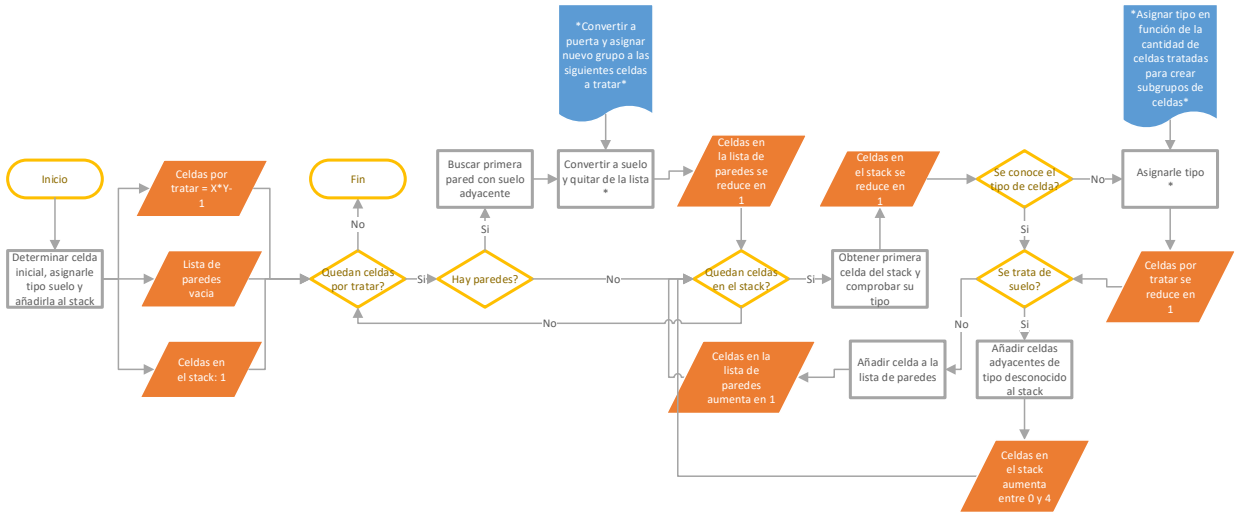

Figura 9-1. Diagrama del funcionamiento del generador de mapas

El generador de mapas empieza creando las paredes externas del mapa y a continuación selecciona una posición aleatoria del mapa para empezar la generación. Esa primera posición se usa también como punto de inicio para el jugador.

A partir de ahí el generador va analizando las celdas contiguas y añadiendo a un stack las que todavía no han sido tratadas, para posteriormente ir tratándolas una a una y asignándoles suelo o pared. En caso de que la celda que está tratando sea una pared se añade a un stack aparte para procesar las paredes (para evitar que el mapa acabe con zonas separadas).

Este paso se repite hasta que no quedan celdas en el stack de celdas por procesar.

Esto genera un mapa de estilo laberíntico con algunas zonas abiertas y con una distribución ligeramente caótica, pero sin desaprovechar ninguna parte del mapa.

Para evitar que los mapas se volvieran monótonos al avanzar con el juego, existen 4 conjuntos distintos de paletas y texturas que se usan con los mapas y los NPC.

Cada conjunto se utiliza durante 8 mapas seguidos y después se cambia al siguiente conjunto.

Las texturas utilizadas para cada conjunto son las siguientes:

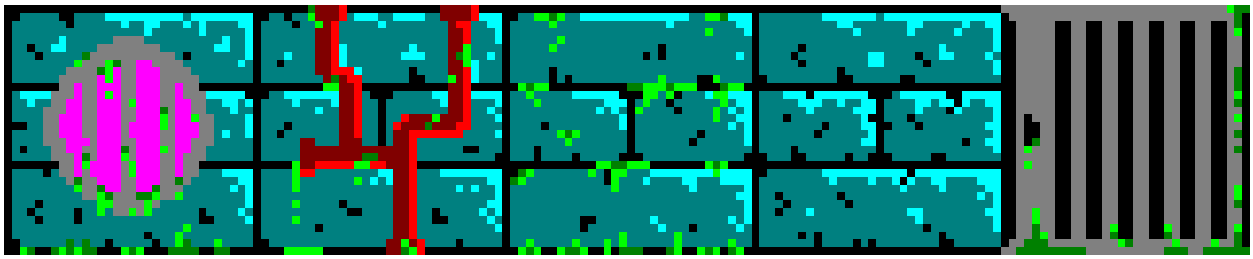

Figura 9-2. Conjunto de texturas de la primera zona

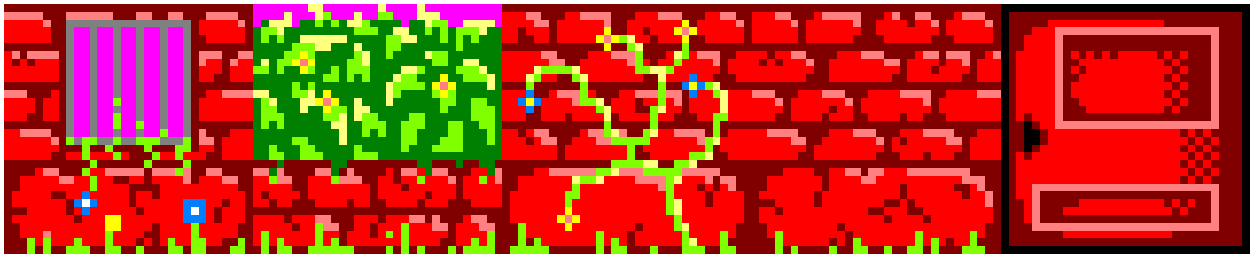

Figura 9-3. Conjunto de texturas de la segunda zona

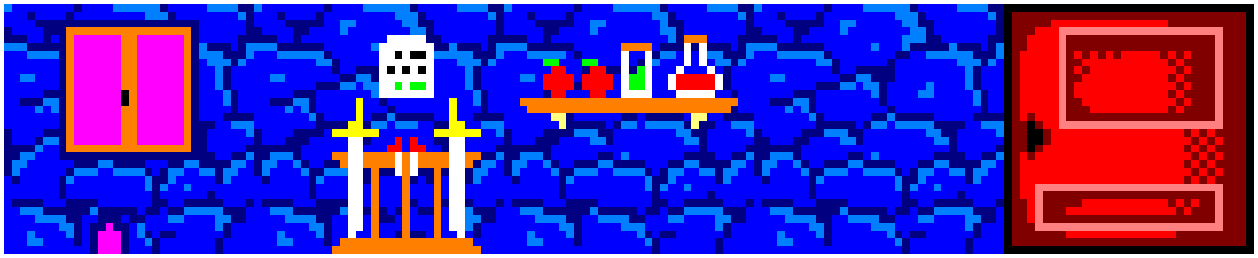

Figura 9-4. Conjunto de texturas de la tercera zona

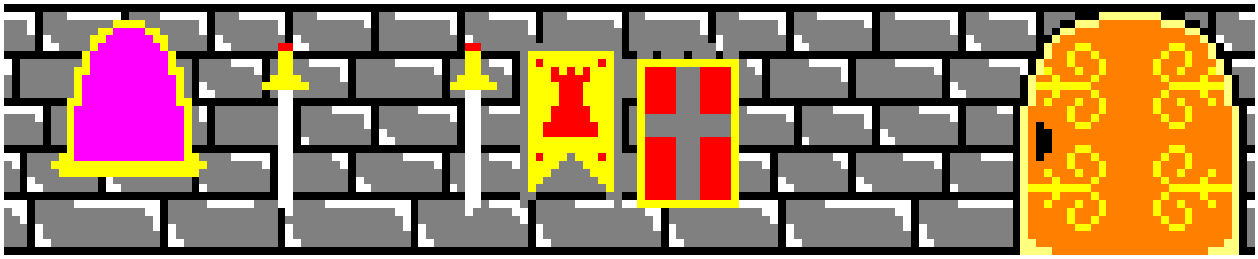

Figura 9-5. Conjunto de texturas de la última zona

9.4.Objetos

Por los mapas aparecen repartidos objetos que el jugador puede usar para mejorar su personaje o para salir de situaciones complicadas.

Además de los objetos que hay en el mapa, los enemigos pueden soltar objetos al morir.

9.4.1.Pociones

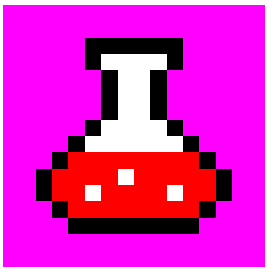

Figura 9-6. Poción

La poción sirve para que el jugador recupere parte de la salud perdida. Se puede usar desde el menú de inventario.

9.4.2.Pergaminos

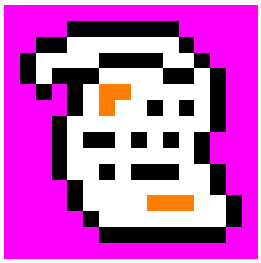

Figura 9-7. Pergamino

El pergamino permite al jugador huir rápidamente de una situación complicada, ya que al usarse teletransporta al jugador a una posición aleatoria del mapa. Se puede usar desde el menú de inventario.

9.4.3.Espadas

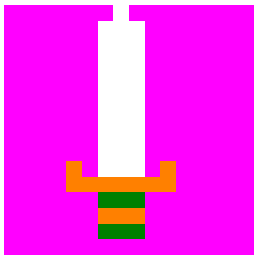

Figura 9-8. Espada

Las espadas que el jugador encuentra por el mapa permiten que pueda ir mejorando a su personaje y poder infringir más daño a los enemigos. Cada espada tiene un valor aleatorio de daño asociado, éste siempre será mayor que el daño actual y estará dentro de un rango dependiente del nivel del juego en el que se encuentre.

9.4.4.Armaduras

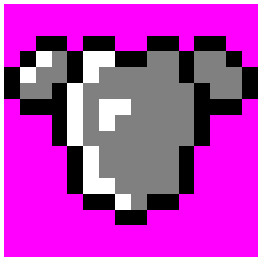

Figura 9-9. Armadura

Las armaduras permiten al jugador mejorar su defensa frente a los ataques de los enemigos. Todas las armaduras que el jugador encuentre son iguales o mejores que la que lleva actualmente, y la mejora que otorga siempre está dentro de un rango que depende del nivel en el que se encuentre.

9.4.5.Llaves

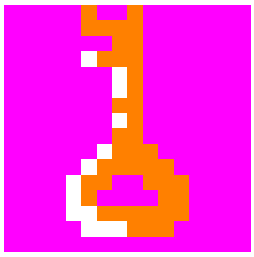

Figura 9-10. Llave

La llave se trata del objeto más importante de cada nivel, es el objeto que necesita el jugador para poder avanzar al siguiente nivel.

Una vez encontrada la llave se debe encontrar la puerta de salida que lleva al siguiente nivel.

9.5.NPCs

Los enemigos que el jugador se puede encontrar escalan en dificultad con el avance del jugador en el juego.

Para cada zona existen 2 tipos de enemigos con comportamientos diferenciados, dependiendo del comportamiento el jugador debe afrontarlos de una manera u otra.

9.5.1.Comportamientos

En un principio se programaron 4 tipos distintos de comportamiento, pero debido al poco espacio disponible en la memoria del Amstrad se tuvo que reducir a 2 tipos distintos.

9.5.1.1.Pasivo

Los enemigos con el comportamiento pasivo ignoraran al jugador cuando lo vean. Pero en caso de que el jugador les ataque, éstos se volverán agresivos y atacarán al jugador.

En cuanto los enemigos con este comportamiento se encuentran débiles tratarán de huir del jugador.

El enemigo que sigue este comportamiento en el juego es la rata, que aparece en todos los niveles.

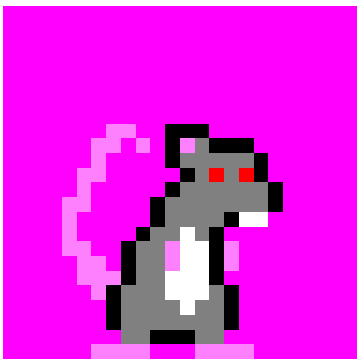

Figura 9-11. Rata (enemigo)

9.5.1.2.Agresivo

Los enemigos agresivos no dudarán en atacar y perseguir al jugador en el momento en que lo tengan a la vista.

Son enemigos muy insistentes que no cesan en intentar matar al jugador y al contrario que los enemigos pasivos, no tratarán de huir cuando estén débiles.

Hay un tipo de enemigo por cada zona que sigue este comportamiento.

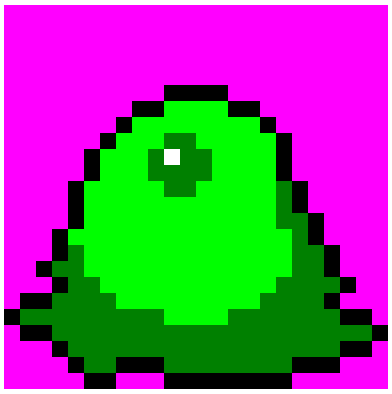

Figura 9-12. Limo (enemigo de la primera zona)

Figura 9-13. Guardia (enemigo de la segunda zona)

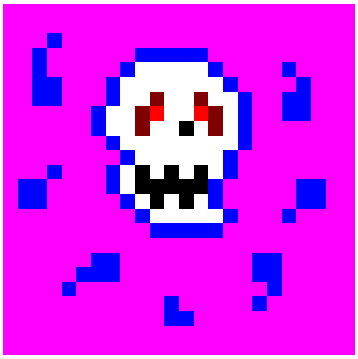

Figura 9-14. Calavera (enemigo de la tercera zona)

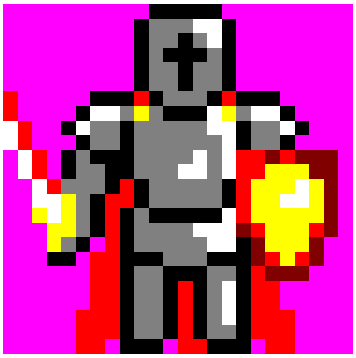

Figura 9-15. Caballero (enemigo de la cuarta zona)

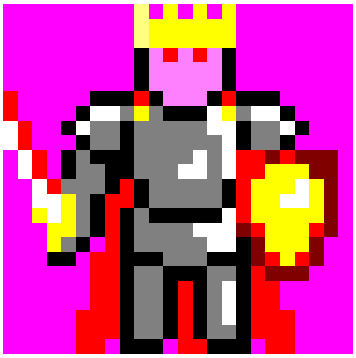

Figura 9-16. Rey (enemigo final del juego)

9.6.Interfaz

La interfaz del juego está principalmente compuesta de menús, ya que todo el juego se controla mediante elementos de menú.

9.6.1.Menús

9.6.1.1.Menú principal

Desde el menú principal se puede empezar nueva partida, acceder al menú de opciones, cargar una partida guardada o acceder a la pantalla de créditos.

Figura 9-17. Menú principal

9.6.1.1.1.Nueva partida

Permite empezar una nueva partida del juego, se empieza en el nivel 1 con un nuevo personaje.

9.6.1.1.2.Cargar partida

Permite cargar una partida previamente guardada. Para cargar la partida se debe introducir una cadena de caracteres válida. En caso de no ser válida no cargará nada.

Para la comprobación, el juego aplica una máscara binaria a la cadena introducida y hace una verificación de suma. Al resultado de la suma se le aplica otra máscara binaria y se compara con un fragmento de la cadena. Si coincide es que la cadena introducida es válida.

```
u8 savegame_checksave(){
    u8 i = SAVEDATA_SIZE-1;
    u8 checksum=0;
    while(i){
        --i;
        checksum+=(saveArray[i]^SAVE_MASK);
    }
    checksum=checksum^CHECKSUM_MASK^SAVE_MASK;
    return (checksum==save.checksum);
}
```

En el código, la variable `save` y la variable `saveArray` apuntan a los mismos datos.

Figura 9-18. Menú de cargar partida

9.6.1.1.3.Opciones

En el menú de opciones se puede cambiar la configuración de la música y de las texturas. Pueden activarse y desactivarse en cualquier momento o desde el menú de opciones del menú principal.

Figura 9-19. Menú de opciones

9.6.1.1.4.Créditos

La pantalla de créditos muestra los créditos del juego.

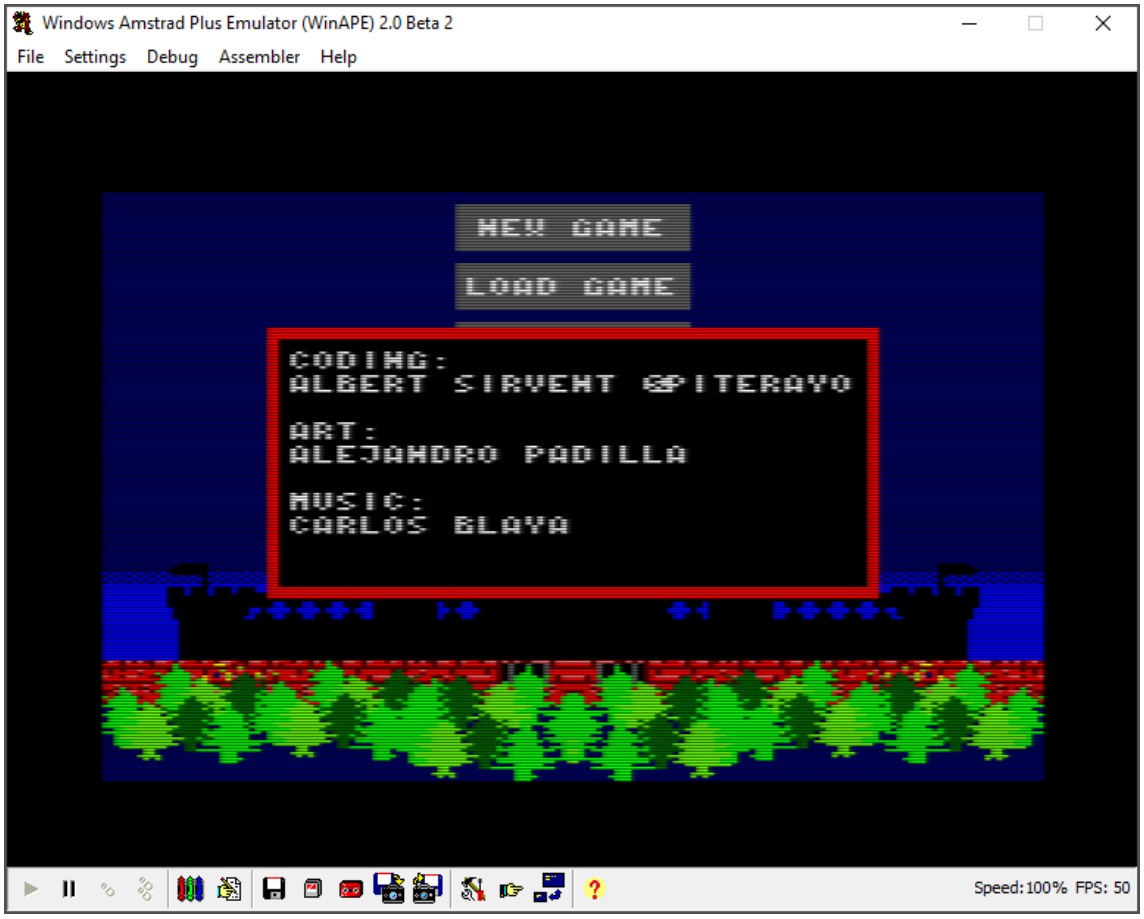

Figura 9-20. Pantalla de créditos

9.6.1.2.Menú de partida

El menú de partida lista las acciones que puede hacer el jugador durante su turno y también da acceso a los menús de inventario y de pausa.

Figura 9-21. Menú de partida

Las acciones de cada botón son las siguientes:

- **Botón de acción:** Permite realizar una acción que dependerá de la situación en la que esté el jugador (atacar si tiene delante a un enemigo o recoger si es un objeto)
- **Botones de movimiento:** Permiten al jugador avanzar o girarse.
- **Botón de espera:** Permite que el jugador pase su turno, esto hará a todos los enemigos avanzar un turno.
- **Botón de inventario:** Permite al jugador acceder a su inventario.
- **Botón de pausa:** Permite pausar el juego y acceder al menú de pausa.

9.6.1.2.1.Menú de inventario

Figura 9-22. Menú de inventario

El menú del inventario permite al jugador utilizar una poción o un pergamino, siempre que tenga unidades disponibles. El uso de uno de estos objetos hace que el turno del jugador acabe.

9.6.1.2.2.Menú de pausa

Figura 9-23. Menú de pausa

En el menú de pausa el jugador puede modificar las opciones del juego, guardar su progreso y salir de la partida o directamente salir al menú principal sin guardar.

Si el jugador decide guardar partida, aparecerá en pantalla una cadena de texto que el jugador debe apuntar y que será la que deba introducir en el menú de cargar partida.

Figura 9-24. Menú de guardar partida

9.6.1.3.Menú de opciones

El menú de opciones del menú de pausa ofrece las mismas opciones que aparecen en el menú principal. Se pueden activar y desactivar tanto la música como las texturas.

Figura 9-25. Menú de opciones

9.6.2.Interfaz de partida

Figura 9-26. Interfaz de partida

La interfaz de la partida ofrece al jugador información sobre su posición, las estadísticas de su personaje, el nivel en el que se encuentra, un registro de lo sucedido en los últimos turnos y le permite ver lo que ve su personaje.

9.6.2.1.Minimapa

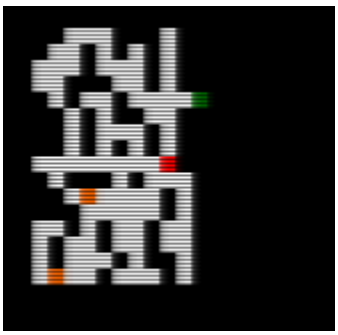

Figura 9-27. Minimapa

El minimapa le da al jugador información de sus alrededores y su propia posición (en rojo). Le marca la posición de los enemigos cercanos (en naranja) y de la salida del mapa (en verde).

Puesto que el mapa es mucho mayor de lo que muestra el minimapa, conforme nos vamos moviendo el minimapa se va actualizando con las zonas de mapa a las que nos vamos acercando.

9.6.2.2.Estadísticas del jugador

Esta parte de la interfaz muestra información sobre el personaje del jugador y el nivel en el que se encuentra.

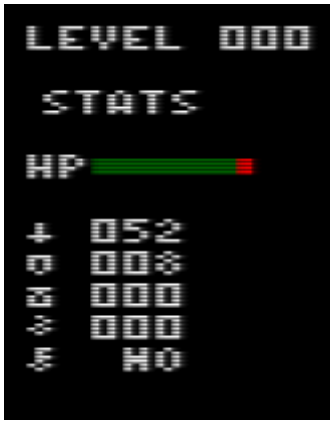

Figura 9-28. Estadísticas del jugador

Los valores mostrados de arriba abajo son los siguientes:

- **Nivel:** Indica en qué nivel del juego se encuentra el jugador.
- **HP:** Indica la salud del personaje, en verde la salud restante y en rojo la faltante.
- El primer símbolo representa una espada, indica el **poder de ataque** del personaje.
- El segundo símbolo representa un escudo, indica la **defensa** del personaje.
- El tercer símbolo representa una poción, indica la **cantidad de pociones** que tiene el jugador en su inventario.
- El cuarto símbolo representa un pergamino, indica la **cantidad de pergaminos** que tiene el jugador en su inventario.
- El ultimo símbolo representa una **llave**, indica si el jugador ha obtenido la llave en el nivel actual.

9.6.2.3.Registro de acciones

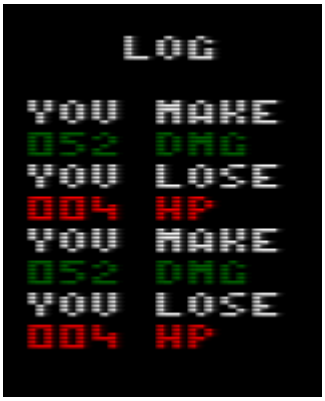

Figura 9-29. Registro de acciones

Sirve para mostrar al jugador un registro de los ataques realizados por él o por los enemigos y el daño realizado por cada uno.

9.6.2.4.Brújula

Figura 9-30. Brújula

Le indica al jugador en qué dirección está mirando.

9.6.2.5.Ventana 3D

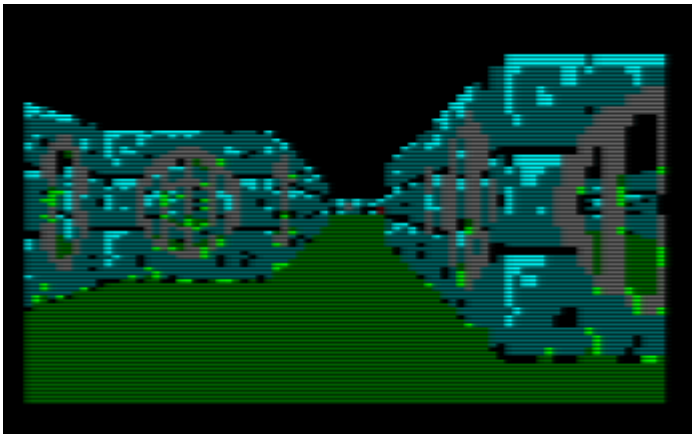

Figura 9-31. Ventana 3D

Esta es la zona donde se ve el renderer en acción, aquí se muestra una visión 3D desde el punto de vista del personaje. Aquí el jugador puede ver en 3D los alrededores del personaje, así como los objetos y enemigos que haya a la vista.

10.Conclusiones

Al inicio de este proyecto no estaba seguro de que lograría conseguir mi objetivo de desarrollar un renderer 3D para una máquina como Amstrad debido a las grandes dificultades técnicas a las que me enfrentaba.

Sin embargo, puedo decir que el resultado obtenido es mucho mejor de lo que esperaba en un principio, he logrado desarrollar un renderer 3D y un videojuego que haga uso de él, con características raramente vistas en juegos de la época del Amstrad, empezando por el 3D, y siguiendo con el sistema de guardado y carga de partidas y la generación de mapas.

Durante la realización del proyecto he necesitado hacer uso de muchos conocimientos adquiridos durante la carrera, pero también he necesitado reforzar otros como la programación a bajo nivel (que, aunque no haya llegado al nivel de ensamblador, sí que he trabajado directamente con la memoria y el alojamiento manual de ésta). También he adquirido nuevos conocimientos y he ganado experiencia como programador, puesto que el reto que suponía este proyecto requería de conocimientos y técnicas de programación considerablemente avanzadas.

El resultado final ha sido un juego en 3D que, a pesar de no ser el más divertido, es jugable cuenta con un alto nivel rejugabilidad gracias a que sus mapas son generados de forma aleatoria.

11.Bibliografía y referencias

[1] **CPCMania:** http://www.cpcmania.com/Docs/Programming/Painting_pixels_introduction_to_video_memory.htm

[2] **Lode's Computer Graphics Tutorial:** Tutorial donde explica el funcionamiento de un renderer de raycast y su implementación. http://lodev.org/cgtutor/raycasting.html#Introduction

[3] **Battlezone:** https://en.wikipedia.org/wiki/Battlezone_(1980_video_game)

[4] **Wayout:** https://en.wikipedia.org/wiki/Wayout

[5] **Wolfenstein 3D:** https://en.wikipedia.org/wiki/Wolfenstein_3D

[6] **Doom:** https://en.wikipedia.org/wiki/Doom_(1993_video_game)

[7] **Quake:** https://es.wikipedia.org/wiki/Quake_(videojuego)

12.Anexos

[1] **Gameplay del juego usando el renderer:** https://www.youtube.com/watch?v=rDEM1u3_3jo

[2] **Repositorio del proyecto:** https://github.com/piterayo/MazeAdventure

[3] **Archivo de GeoGebra usado para la maquetación del renderer:** https://drive.google.com/file/d/0Bzj8--X5OLbAdnZiM1dBWll0d3M/view?usp=sharing